Computing the Least Fix-point Semantics of Definite Logic Programs Using BDDs

Frédéric Besson, Thomas Jensen, Tiphaine Turpin

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1939 – November 23, 2009

Computing the Least Fix-point Semantics of Definite Logic Programs Using BDDs

Frédéric Besson[*] , Thomas Jensen[**] , Tiphaine Turpin[***]

Abstract: We present the semantic foundations for computing the least fix-point semantics of definite logic programs using only standard operations over boolean functions. More precisely, we propose a representation of sets of first-order terms by boolean functions and a provably sound formulation of intersection, union, and *projection* (an operation similar to *restriction* in relational databases) using conjunction, disjunction, and existential quantification. We report on a prototype implementation of a logic solver using Binary Decision Diagrams (BDDs) to represent boolean functions and compute the above-mentioned three operations. This work paves the way for efficient solvers for particular classes of logic programs *e.g.*, static program analyses, which leverage BDD technologies to factorise similarities in the solution space.

Key-words: Semantics, binary decision diagrams, logic programs

---

Calcul de la sémantique de plus petit point-fixe de programmes logiques définis en utilisant des BDDs

Résumé : *Nous présentons les fondements sémantiques nécessaires pour calculer la sémanique de plus petit point-fixe de programmes logiques définis, en utilisant uniquement des opérations standard sur les fonctions booléennes. Plus précisément, nous proposons une représentation d'ensembles de termes du premier ordre par des fonctions booléennes, et une formulation (prouvée correcte) de l'intersection, l'union et la* projection *(une opération similaire à la* restriction *dans les bases de données relationnelles) qui utilise la conjonction, la disjonction et la quantification existentielle. Nous rapportons les résultats d'un prototype d'implémentation d'un solveur logique qui utilise des diagrammes de décision binaires (BDDs) pour représenter les fonctions booléennes et calculer ces trois opérations. Ce travail ouvre la voie à des solveurs efficaces pour des classes particulières de programmes logiques, par exemple, des analyses statiques de programmes, qui utilisent la technologie des BDDs pour factoriser les similarités dans l'espace de solutions.*

Mots clés : *Sémantique, diagrammes de décision binaires, programmes logiques*

---

[*] INRIA Rennes - Bretagne Atlantique
[**] IRISA/CNRS
[***] INRIA Rennes - Bretagne Atlantique

Contents

1 Introduction

Since their introduction by Bryant, Binary Decisions Diagrams [7] (BDDs) have proved to be a very efficient data-structure for representing large sets and relations, provided that sufficient similarities exist which can be factorised. They have been successfully used in various areas of computer science as a way to obtain scalability. Success stories include hardware verification [9], symbolic model checking [8], software model checking [2].

However, BDD is a low-level data type, and expressing the solution of a problem in terms of BDDs requires considerable work. Hence, higher-level languages have been proposed, which make the specification of a problem easier and can be compiled to BDD operations. Examples include the Ever [17] language, the work of Iwaihara and Inoue [18], the Crocopat [5] system, and `bddbddb` [25] which all use relations over finite domains as their basic data structure. In particular the input language for `bddbddb` is DATALOG which is basically the subset of PROLOG obtained by excluding all first-order terms except constants and variables.

The work presented here is, to the best of our knowledge, the first attempt to introduce first-order terms in a BDD-based logic solver, thus obtaining the expressive power of PROLOG. More precisely, the contribution of this report consists in the algorithmic and semantic foundations for such a solver, *i.e.*, a boolean function-based operational semantics which computes the least fix-point of pure *definite* (*i.e.*, without negation) logic programs.

Overview The first step toward this goal will be the identification of three basic operations on sets of terms, namely, union, intersection, and *projection* (analogous to a *restriction*), which allow the expression of the least fix-point semantics of logic programs. Then we will address the computation of those operations on a boolean function-based representation of sets of terms. Intuitively, the set of instances of a single term will be represented by a conjunction of atoms which describe the function symbols and variable equalities defining this term. Accordingly, a set of terms (which, in our case, is always the set of instances of a finite number of terms) is represented by a disjunction of such conjunctions (*i.e.*, a formula in Disjunctive Normal Form). Therefore, the union of sets of terms is naturally implemented by a disjunction of boolean formulas. The intersection is also correctly implemented by a conjunction, but in an implicit and "lazy" way, as a constructive implementation of the intersection of the instances of two finite sets of terms amounts to unifying the terms pairwise. Then, we will show that the projection a set of terms with respect to a given *path* (which identifies a position in the term) can be computed by performing an existential quantification with respect to a set of atoms which are about this path, but this is only sound for a DNF formula whose disjuncts satisfy a particular *closure* property (mainly, the absence of *implied* atoms). This hypothesis is met by applying an appropriate *closure* procedure to the result of conjunction operations, which in fact implements the actual unification process associated with the intersection of sets of terms. The interest of performing the unification using formulas is that we can have a representation of those (namely, BDDs), whose size in not related to the number of disjuncts of the DNF (which correspond to the most general terms satisfying the formula) and therefore the complexity of unifying two formulas in a single process can be much lower than unifying all the corresponding pairs of terms independently.

The most challenging difficulty in this work is the treatment of the term equalities implicitly represented by multiple occurrences of a variable. This is what makes the elaborate closure property necessary, and the soundness proof for the projection operation subtle. We solve this problem in the general case, in particular without requiring that the programs be *range-restricted*.

Organisation of the report The definition of the least-fix-point semantics of logic programs and its expression with set operations are described in Section 2. Section 3 defines precisely the boolean function-based encoding of sets of terms. Section 4 is dedicated to the implementation of the *projection* operation on *closed* boolean functions and its soundness proof. Section 5 describes the closure procedure which provides an implementation of intersection which is compatible with that of projection. Section 6 discusses various optimisations to the formal development presented here that we have used in a prototype implementation, and briefly discusses preliminary experimental results. Section 7 mentions related topics with respect to the representation of sets of terms. In the conclusion (Section 8) we set our work back in the context of static program analysis and detail other possible further work.

2 Evaluating logic programs using set operations

We briefly recall the definition of the least fix-point semantics of logic programs, and we show how to express the immediate consequence operator T_c for a clause using basic set operations.

2.1 Least fix-point semantics

The least fix-point, or bottom-up semantics [19] of a logic program is defined as the set of facts that are recursively deducible from its clauses. It is computed by repeatedly applying the *immediate consequence* operator associated with each clause, which is defined as follows. Let c be a clause $h :- b_1, \ldots, b_n$. We assume that h and $b_1, \ldots, b_n$ are terms (in other words, predicate names are assumed to be particular function symbols) thus the semantics of the program is just as set of terms. The immediate consequences of c for a set S of terms are given by the operator T_c defined by:

$$T_c(S) = \{h\sigma \mid \forall i \leq n \ b_i\sigma \in S\}.$$

2.2 Computing T_c with set operations

The computation of $T_c(S)$ involves three steps. First for each premise b_i, we compute the set of matching substitutions $\{\sigma \mid b_i\sigma \in S\}$. Each substitution σ is represented by a term $subst(\sigma(x_1), \ldots, \sigma(x_p))$ where $x_1, \ldots, x_p$ are the variables appearing in clause c, and *subst* is a particular function symbol. Then we intersect those sets. Finally, we apply the resulting set of substitutions to the head. For the first and last steps, we rely on special terms which represent the notion of instantiation: for each term u, we consider the term $\langle u, subst(x_1, \ldots, x_p)\rangle$ (where $\langle\cdot,\cdot\rangle$ is a particular function symbol). This term has the property that the set of its instances represents the set of pairs $\{\langle u\sigma, \sigma\rangle\}$. Given this property, the set of matching substitution for a premise b_i can be computed as:

$$\{\sigma \mid b_i\sigma \in S\} = \pi_{snd}(\text{inst}(\langle b_i, subst(x_1, \ldots, x_p)\rangle) \cap (S \times \mathcal{T}(\mathcal{F}, \mathcal{X}))).$$

In this expression, π_{snd} represent the projection of a set of pairs onto their second components, the function inst returns the set of instances of a term, and $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of all terms. Similarly, the application of a set of substitutions S to the head is given by:

$$\{h\sigma \mid \sigma \in S\} = \pi_{fst}(\text{inst}(\langle h, subst(x_1, \ldots, x_p)\rangle) \cap (\mathcal{T}(\mathcal{F}, \mathcal{X}) \times S)).$$

Thus, we obtain an implementation of the operator T_c using the operations $\cap$, π_{fst}, π_{snd}, $\times$, and inst. With the addition of $\cup$, this allows us to express the least fix-point semantics of a logic program. In the remaining of this report, we will show how to compute those operations on a boolean formula-based representation of sets of terms. As a minor variation, instead of $\times$, π_{fst}, and π_{snd}, our formalisation will use an existential quantification operator $\exists p$ (with p is a *path* such as *fst* or *snd*) which plays the same role, and is informally defined by $\exists snd\ S = \pi_{fst}(S) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$ (and the other way around).

3 Representing sets of terms with boolean functions

In this section we define a representation of sets of first-order terms by boolean functions, and show that the union and intersection of sets of terms, as well as the set of instances of a single term, can be expressed on those boolean functions.

3.1 Terms and boolean functions

We first recall standard definitions about first-order terms and boolean functions.

3.1.1 First-order terms

Let $\mathcal{F}$ be a ranked alphabet and $\mathcal{X}$ a set of variables. We write $\mathcal{T}(\mathcal{F}, \mathcal{X})$ for the set of first-order terms on $\mathcal{F}$ and $\mathcal{X}$. The application of a substitution σ to a term u is denoted by $u\sigma$, and the substitution which maps the variable x to the term u is denoted by $[u/x]$. For all $S \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$, we define $\text{inst}(S) \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$ as the set of instances of the terms of S:

$$\text{inst}(S) = \{u\sigma \mid u \in S, \sigma \in \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})\}$$

and we let $\text{inst}(u) = \text{inst}(\{u\})$.

3.1.2 Boolean functions

Let $\mathcal{A}$ be a set of boolean variables. We call *formula* a boolean function over $\mathcal{A}$ which only depends on a finite subset of $\mathcal{A}$. We write $\mathcal{B}(\mathcal{A})$ for the set of formulas on $\mathcal{A}$:

$$\mathcal{B}(\mathcal{A}) = \ \{\phi : (\mathcal{A} \to \text{boolean}) \to \text{boolean} \mid |\text{support}(\phi)| < \infty\}$$

where the support of a boolean function is defined in the obvious way, *i.e.*,

$$\text{support}(\phi) = \{a \in \mathcal{A} \mid \exists v, v' \ \phi(v) \neq \phi(v') \ \wedge \ \forall a' \neq a \ v(a') = v'(a')\}.$$

Elements of $\mathcal{B}(\mathcal{A})$ can be denoted by finite propositional formulas over $\mathcal{A}$, which are interpreted up to propositional equivalence. In particular, we will sometimes assume that formulas are in *disjunctive normal form*, which we define as a disjunction of non-false conjunctions of possibly negated atoms (minimality of the disjuncts is not required).

For a formula $\phi \in \mathcal{B}(\mathcal{A})$, a valuation $v : \mathcal{A} \to$ boolean, and a positive atom $a \in \mathcal{A}$, we write $v \in \phi$ if $\phi(v)$ is true and $a \in v$ if $v(a)$ is true.

3.2 Paths, term formulas and their interpretation

Our aim is to represent sets of terms by boolean functions. The core idea is to use formulas whose atoms describe the function symbols that are reached by following each possible path in the terms. Additional equality atoms will also be required to represent variable equality in terms.

Example 1 *Consider the term $u = f(x, g(x))$. Then u will be represented by the formula*

$$\epsilon \mapsto f \ \wedge \ f.2 \mapsto g \ \wedge \ f.1 = f.2.g.1.$$

3.2.1 Paths

To identify the position of the sub-terms of (sets of) terms, we use a notion of *path*. A path is an alternating sequence of function symbols and integers where each integer is the index of an argument of the preceding function symbol. We write P for the set of paths:

$$P = \{f_1.p_1. \ldots .f_n.p_n \mid n \geq 0, \forall i \leq n \ f_i \in \mathcal{F} \ \wedge \ 1 \leq p_i \leq \text{arity}(f_i)\}.$$

The empty path is denoted by ϵ. We write $u(p)$ for the sub-term of u obtained by following the path p, if any, and $p \in u$ if $u(p)$ is defined. We also write $u[u'/p]$ for the term u where the sub-term reached by path p has been replaced by u'. The notation $\mathcal{C}_p(u) = f$ expresses that $u(p)$ is defined, is not a variable, and that its top-most function symbol is f.

Example 2 *For the term $u = f(x, g(x))$ of Example 1, the paths which are defined in u are $\epsilon, f.1, f.2, f.2.g.1$, and we have for example $u(f.1) = x$ and $\mathcal{C}_{f.2}(u) = g$.*

3.2.2 Formulas

We consider the following set $\mathcal{A}$ of positive atoms a:

$$\mathcal{A} \ni a ::= p \mapsto f \mid p = p'$$

where $p, p' \in P$ and $f \in \mathcal{F}$. Equality atoms are defined up to commutation of their parameters, so that $p = p'$ and $p' = p$ are the same positive atom. Intuitively, the first kind of atoms match the outer-most function symbol of a sub-term, while the second kind force the equality of two sub-terms. In the following, we always consider formulas on $\mathcal{A}$, and we write $\mathcal{B}$ for the set of formulas (*i.e.*, $\mathcal{B} = \mathcal{B}(\mathcal{A})$).

3.2.3 Meaning

Formulas are interpreted as sets of terms in the following way. The meaning of positive atoms is given by a satisfaction relation $\models$ between terms and atoms defined as follows:

$$\begin{array}{lcll} u & \models & p \mapsto f & \text{iff} \quad p \in u \ \wedge \ \mathcal{C}_p(u) = f \\ u & \models & p = p' & \text{iff} \quad p \in u \ \wedge \ p' \in u \ \wedge \ u(p) = u(p'). \end{array}$$

As formulas are defined as boolean functions, the meaning of formulas is first defined in terms of valuations. A more intuitive expression is provided by Lemma 1. The meaning of a formula ϕ is defined as:

$$[\![\phi]\!] = \{u \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \mid \exists v \in \phi \ \forall a \in \mathcal{A} \ u \models a \iff a \in v\}.$$

Lemma 1 *For any positive atom a, and any formulas ϕ and ϕ', the following holds:*

$$\begin{array}{lcl} [\![a]\!] & = & \{u \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \mid u \models a\} \\ [\![\neg \ \phi]\!] & = & \mathcal{T}(\mathcal{F}, \mathcal{X}) \setminus [\![\phi]\!] \\ [\![\phi \ \wedge \ \phi']\!] & = & [\![\phi]\!] \cap [\![\phi']\!] \\ [\![\phi \ \vee \ \phi']\!] & = & [\![\phi]\!] \cup [\![\phi']\!]. \end{array}$$

Proof 1 *This follows from the definitions.*

This lemma shows that the usual set operations on sets of terms may be expressed straightforwardly on the formula representation using their logical equivalents.

3.2.4 Representing the instances of a single term

The set of instances of single term can be represented by a formula as shown by Lemma 2. This implies that boolean functions are expressive enough to represent any set of terms obtained as the set of instances of a finite number of terms.

Lemma 2 *Let u be a term. Define ϕ_u as*

$$\phi_u = \bigwedge_{\mathcal{C}_p(u)=f} p \mapsto f \ \wedge \bigwedge_{\substack{p \neq p' \\ u(p)\in\mathcal{X} \\ u(p)=u(p')}} p = p'.$$

Then $[\![\phi_u]\!] = \mathrm{inst}(u)$.

Proof 2 *This follows from the definitions.*

Remark: The formula of Example 1 corresponds precisely to ϕ_u.

4 Projecting sets of terms

This Section defines the projection of a set of terms with respect to a path and shows that it can be computed by an existential quantification of a formula, provided an additional *closure* assumptions on this formula.

4.1 Introducing the projection

The purpose of the projection operator is to abstract a set of terms with respect to a given path. This is very similar to the projection operations π_{fst} and π_{snd} described in Section 2, except that the sub-terms under the projected path are not removed but replaced by every possible term.

4.1.1 Projection of a set of terms

The projection $\exists p\ u$ of a term u with respect to a path p is defined as the set of terms obtained by substituting any term in $\mathcal{T}(\mathcal{F},\mathcal{X})$ for the path p (if it appears) in u. Formally:

$$\exists p\ u = \begin{cases} \{u[u'/p] \mid u' \in \mathcal{T}(\mathcal{F},\mathcal{X})\} & \text{if } p \in u \\ \{u\} & \text{otherwise.} \end{cases}$$

The projection is extended to sets of terms in the obvious way:

$$\exists p\ S = \{\exists p\ u \mid u \in S\}.$$

4.1.2 The projection problem

The intuition for implementing the projection on formulas is the same as for the other operations: compute the projection on a path p by projecting (*i.e.*, existentially quantifying) on the atoms which involve p or a suffix of p. Formally, the set $\mathcal{A}_p$ of atoms implying a path p is defined by

$$\mathcal{A}_p = \{p.p' \mapsto f \in \mathcal{A}\} \cup \{p.p' = p'' \in \mathcal{A}\}.$$

However, the formulas in $\mathcal{B}$ are too general to allow this simple translation, and thus need to be refined. Example 3 illustrates the projection problem.

Example 3 *Consider the ranked alphabet $\mathcal{F} = \{f/3\}$ and the two formulas $\phi = \epsilon \mapsto f \ \wedge\ f.1 = f.2 \ \wedge\ f.2 = f.3$ and $\psi = \phi \ \wedge\ f.1 = f.3$. We have*

$$[\![\phi]\!] = [\![\psi]\!] = \{f(u,u,u) \mid u \in \mathcal{T}(\mathcal{F},\mathcal{X})\}.$$

Now suppose that we want to project this set of terms on the path $f.2$. We get

$$\exists f.2\ [\![\phi]\!] = \{f(u,u',u) \mid u,u' \in \mathcal{T}(\mathcal{F},\mathcal{X})\}.$$

On the formula side, we proceed by quantifying existentially with respect to $\mathcal{A}_{f.2} = \{f.2 = f.1, f.2 = f.3, \ldots\}$, and we obtain $\exists\mathcal{A}_{f.2}\ \phi = \epsilon \mapsto f$ and $\exists\mathcal{A}_{f.2}\ \psi = \epsilon \mapsto f \ \wedge\ f.1 = f.3$. We observe that

$$[\![\exists\mathcal{A}_{f.2}\ \phi]\!] = \{f(u,u',u'') \mid u,u',u'' \in \mathcal{T}(\mathcal{F},\mathcal{X})\}$$

and thus $[\![\exists\mathcal{A}_{f.2}\ \phi]\!] \neq \exists f.2\ [\![\phi]\!]$, but $[\![\exists\mathcal{A}_{f.2}\ \psi]\!] = \exists f.2\ [\![\psi]\!]$. The reason is that the atom $f.1 = f.3$ which is implied in ϕ is not preserved by the existential quantification. In ψ however, this atom is explicit, and thus preserved.

Given this example we can make two remarks, which sketch the main theoretical results of this work. First, we can apply the existential quantification to a conjunctive formula to compute the projection provided the formula is "closed" with respect to implicit atoms. This is formalised in the remaining of this section, in two steps: after defining the notion of closed conjunction in Section 4.2, we state in Section 4.3 a key result giving a constructive expression of the meaning of closed conjunctions ; then this result is used in Section 4.4 to prove the soundness of existential quantification-based projection for (disjunctions of) closed conjunctions. Second, the conjunction operator on formulas, used to implement intersection, does not preserve the closure property (the formula ϕ can be obtained trivially as the conjunction of the two closed conjunctions $\epsilon \mapsto f \ \wedge \ f.1 = f.2$ and $\epsilon \mapsto f \ \wedge \ f.2 = f.3$). Therefore, closure must be restored after each conjunction operation. This constitutes the main topic of Section 5, with Section 5.3 gathering those results in a boolean function-based domain which is closed under union, intersection, and projection.

4.2 Closed conjunctions

The notion of closed conjunctions is defined by a collection of constraints, which are presented in turn.

4.2.1 Conjunctions

In the following, we will call *conjunction* a non-false formula which can be expressed as a conjunction of (possibly negated) atoms. This representation is obviously unique up to symmetry (assuming distinct conjuncts), and cannot contain both a positive atom and its negation. We write $a \in \phi$ (respectively $\neg\, a \ \in \ \phi$) if a (respectively $\neg\, a$) is one of the conjuncts of ϕ.

4.2.2 Prefix-closed conjunctions

We say that a conjunction ϕ is *prefix-closed* if for all p, f, i, p', and f',

$$(p.f.i \mapsto f' \ \in \ \phi \ \vee \ p.f.i = p' \ \in \ \phi) \implies p \mapsto f \ \in \ \phi$$

(remember that equality atoms are defined up to commutation, so that $p.f.i = p$ is identical to $p = p.f.i$). In other words, for any positive atom involving a path $p.f.i$ occuring in ϕ, the positive atom $p \mapsto f$ must also occur. The purpose of this definition is to make explicit the implied constraints on the prefixes of paths appearing in a positive atom, namely, the fact that $(u \ \models \ p \mapsto f) \implies p \in u$ and $(u \ \models \ p = p') \implies p, p' \in u$.

4.2.3 Positive conjunctions

The second constraint restricts the use of negation to a single case : only function symbol atoms may be negated, provided that a positive atom i with the same path is present. Formally, a conjunction ϕ is *positive* if

$$\forall a \in \mathcal{A} \ (\neg\, a) \in \phi \implies \exists p, f, f' \ a = p \mapsto f \ \wedge \ p \mapsto f' \in \phi.$$

4.2.4 Deduction rules

The last (and essential) constraint is expressed by a set of *rules* that a conjunction must *satisfy* to make sure that it has no implied atoms (an atom is implied by a conjunction if it is satisfied by every term which satisfies the conjunction). A rule is composed of a set of atoms h_i which are called the hypotheses and are interpreted in a conjunctive sense, and a set of conclusions c_i interpreted in a disjunctive sense (in practice however we only consider rules with at most one conclusion). The empty set of conclusions is denoted by $\bot$, and rules are noted

$$\frac{h_1 \ \ldots \ h_n}{c_1, \ldots, c_m}, \qquad \frac{h_1 \ \ldots \ h_n}{\bot}.$$

We define the following five rules schemes:

$$\textsc{mat}_1 \frac{p = p' \quad p \mapsto f}{p' \mapsto f} \qquad \textsc{mat}_2 \frac{p = p' \quad p \mapsto f}{p.f.i = p'.f.i} \text{ where } 1 \leq i \leq \text{arity}(f)$$

$$\textsc{trans} \frac{p = p' \quad p' = p''}{p = p''} \qquad \textsc{cycle} \frac{p = p.p'}{\bot} \text{ where } p' \neq \epsilon$$

$$\textsc{conflict} \frac{p \mapsto f \quad p \mapsto f'}{\bot} \text{ where } f \neq f'$$

We first give the intuition behind those rules (whose precise relation with the meaning of formulas is stated in Properties 1 and 2). The rules $\textsc{mat}_1$ and $\textsc{mat}_2$ have a *materialisation* purpose: they are used to introduce the consequences of sub-term

equalities. The rule TRANS expresses the transitivity of equality, and the rules CYCLE and CONFLICT detect inconsistent situations.

We denote by $\mathcal{R}$ the (infinite) set of rules generated by the above schemes. In the following, we call *rule* an element of $\mathcal{R}$. An important property of those rules is that they are both sound and complete with respect to the meaning of formulas. This is stated in Properties 1 and 2.

Property 1 (Soundness) *Let u be a term and $R \in \mathcal{R}$. If u satisfies all the hypotheses of R then u satisfies at least one of the conclusions of R.*

Proof 3 *This follows from the definitions.*

Property 2 (Completeness) *Let ϕ be a conjunction and ψ a disjunction of positive atoms. If $[\![\phi]\!] \subseteq [\![\psi]\!]$, then $\phi \vdash_{\mathcal{R}} \psi$, where the deduction relation $\vdash_{\mathcal{R}} \in \mathcal{P}(\mathcal{A}) \times \mathcal{P}(\mathcal{A})$ is defined in the standard way. In particular, if $[\![\phi]\!] = \emptyset$ then $\phi \vdash_{\mathcal{R}} \bot$.*

Idea of the proof. The relation $\vdash_{\mathcal{R}}$ can be shown to simulate a unification algorithm. The result follows from the functional correctness of unification and its termination which mainly relies on so-called *occurs checks*, whose equivalent here is the rule CYCLE.

4.2.5 Rule satisfaction

The fact that a conjunction ϕ satisfies a rule R (which we write $\phi \models R$) is defined as follows:

$$\phi \models \frac{h_1 \ \ldots \ h_n}{c_1 \ \ldots \ c_m} \quad \text{iff} \quad (\forall i \leq n \ h_i \in \phi) \implies (\exists j \leq m \ c_j \in \phi).$$

4.2.6 Closed conjunctions

We can now define closed conjunctions using the above constraints. A conjunction ϕ is *closed* if a) ϕ is prefix-closed, b) ϕ is positive, and c) ϕ satisfy every rule in $\mathcal{R}$.

4.3 A characterisation of the meaning of closed conjunctions

We complete the presentation of closed conjunctions by proving a fundamental property about their meaning. In Lemma 3 we define, for any closed conjunction ϕ, a term u_ϕ whose set of instances is equal to the meaning of ϕ. In other words, u_ϕ is the most general term satisfying ϕ. A particularly important consequence of this result is that closed conjunctions are satisfiable. This satisfiability feature is the key to the proof of Lemma 4 (which establishes the soundness of projection): in this lemma, given a term u satisfying $\exists \mathcal{A}_p \ \phi$ (here, $\exists \mathcal{A}_p \ \phi$ is the projection of ϕ onto some path p), we have to modify u (by substituting some appropriate term to the path p) such as to satisfy ϕ, thus ensuring that $u \in \exists p \ [\![\phi]\!]$.

Lemma 3 *Let ϕ be a closed conjunction. Then there exists a term u_ϕ such that $[\![\phi]\!] = \text{inst}(u_\phi)$.*

Proof 4 *We give a constructive definition of u_ϕ, and prove that $[\![\phi]\!] = \text{inst}(u_\phi)$.*

First, we remark that, because of rule TRANS, *$p = p' \in \phi$ is an equivalence relation over paths. We let $\overline{p}$ be the equivalence class of a path p. Now, for every path p, we define a term u_p and show that for every atom $(\neg)\ p.p' \mapsto f$ in ϕ (either positive and negative), $u_p \ \models \ (\neg)\ p' \mapsto f$. The definition and proof are done by induction on the set of paths p' such that some positive atom $p.p' \mapsto f$ belongs to ϕ:*

- *If there exist (possibly negated) atoms of the form $(\neg)\ p \mapsto f$ in ϕ, then as ϕ is positive, one of those atoms is positive, and by the rule* CONFLICT, *it is unique. Let $u_p = f(u_{p.f.1}, \ldots, u_{p.f.n})$ where $n = \text{arity}(f)$. By induction hypothesis, for all $i \leq n$, $u_{p.f.i}$ satisfies all the atoms $(\neg)\ p' \mapsto f'$ such that $(\neg)\ p.f.i.p' \mapsto f' \in \phi$, thus u_p satisfies $(\neg)\ f.i.p' \mapsto f'$ for the same p', f'. Furthermore, u_p obviously satisfies all the atoms $(\neg)\ \epsilon \mapsto f'$ such that $(\neg)\ p \mapsto f' \in \phi$. Finally, as ϕ is prefix-closed, there cannot be any positive atom $p.f'.i.p' \mapsto f''$ with $f' \neq f$ in ϕ. Thus, u_p satisfies all the atoms $(\neg)\ p' \mapsto f'$ such that $(\neg)\ p.p' \mapsto f' \in \phi$.*
- *Otherwise we let u_p be the variable $x_{\overline{p}}$ and as ϕ is prefix-closed, the result follows by the same argument.*

We conclude that the term $u_\phi = u_\epsilon$ satisfies all the atoms $(\neg)\ p \mapsto c$ in ϕ. Now we have to deal with path equalities. We observe that, because u satisfies ϕ and ϕ is prefix closed, for every equality $p = p' \in \phi$, $p \in u$ and $p' \in u$. We then prove by induction on $u(p)$ that for every such equality, $u(p) = u(p')$. First, because of rule MAT_1 *and since u satisfies the constraints of ϕ of the form $(\neg)\ p \mapsto c$, we know that $u(p)$ is a variable if and only if $u(p')$ is, and otherwise $\mathcal{C}_p(u) = \mathcal{C}'_p(u)$. We prove the two cases in turn:*

- *If $u(p)$ and $u(p')$ are the variables $x_{\overline{p}}$, $x_{\overline{p'}}$, then $\overline{p} = \overline{p'}$ by definition thus the atom is satisfied.*
- *If $u(p) = f(u_1, \ldots, u_n)$ and $u(p') = f(u'_1, \ldots, u'_n)$ then the rule* MAT$_2$ *ensures that for all $i \leq n$, $p.f.i = p'.f.i \in \phi$. By induction hypothesis we deduce that $u(p.f.i) = u(p'.f.i)$ and the result follows.*

Therefore $u_\phi \models \phi$.

Finally, we must prove that every term u satisfying ϕ is an instance of u_ϕ. By definition, for every equality $p = p'$ in ϕ, we know that $p \in u$, $p' \in u$, and $u(p) = u(p')$. We denote this term by $u(\overline{p})$ (where $\overline{p}$ is again the equivalence class of p with respect to the relation $p = p' \in \phi$). It follows by induction on u_ϕ that $u = u_\phi[u(\overline{p})/x_{\overline{p}}]$.

Remark about the proof of Lemma 3: *The fact that closed conjunctions satisfy the rule* CYCLE *has not been used in the proof (so the closure hypothesis is slightly stronger than necessary). However, this rule is required for completeness (Property 2), and this property will be used in Section 5.2.*

4.4 Projection

Using the characterisation of the meaning of closed conjunctions provided by Lemma 3, we prove the soundness of using existential quantification on formulas to express the projection of sets of terms, for a restricted set of formulas that are disjunctions of closed conjunctions. Lemma 5 establishes the result, the essential step being proved in Lemma 4.

Lemma 4 *Let p be a path and ϕ be a closed conjunction. Define u_ϕ and $u_{(\exists \mathcal{A}_p\ \phi)}$ as in Lemma 3. Then*

$$\mathrm{inst}(\exists p\ u_\phi) = \mathrm{inst}(u_{(\exists \mathcal{A}_p\ \phi)}).$$

Remark: The statement of Lemma 4 implicitely assumes that $\exists \mathcal{A}_p\ \phi$ is a closed conjunction. The three constraints which define the closure of this formula are easily checked from the definition of $\mathcal{A}_p$.

Proof 5 *There are two cases.*

- *If $p \notin u_\phi$, then it is easy to see that $u_\phi = u_{(\exists \mathcal{A}_p\ \phi)}$ and the results follows.*
- *Otherwise, we define a substitution σ such that*

$$\exists p\ u_\phi = \{u_{(\exists \mathcal{A}_p\ \phi)}\sigma[x_p \mapsto u] \mid u \in \mathcal{T}(\mathcal{F}, \mathcal{X})\} \tag{1}$$

where x_p is a particular variable appearing in the range of σ. First, we remark that, because ϕ satisfies TRANS*, the equivalence classes for the relation $p' = p'' \in \exists \mathcal{A}_p\ \phi$ are the same ones as for the relation $p' = p'' \in \phi$, except that the class $\overline{p}$ of the latter may be splited into two in the former. More precisely, if we use the notations $\overline{\cdot}^{\exists \mathcal{A}_p\ \phi}$ and $\overline{\cdot}^{\phi}$ for those equivalence classes, then either $\overline{p}^{\exists \mathcal{A}_p\ \phi} = \overline{p}^{\phi} = \{p\}$ or $\overline{p}^{\phi}$ is splited into $\{p\}$ and $\overline{p}^{\phi} \setminus \{p\}$. We thus define σ by*

$$\begin{array}{lcll} \sigma(x_{\overline{p'}^{\exists \mathcal{A}_p\ \phi}}) & = & x_{\overline{p'}^{\phi}} & \text{for all } p' \neq p \text{ such that } p = p' \notin A \\ \sigma(x_{\overline{p}^{\phi} \setminus \{p\}}) & = & x_{\overline{p}^{\phi}} & \text{if } \overline{p}^{\phi} \setminus \{p\} \neq \emptyset \\ \sigma(x_{\{p\}}) & = & x_p & \text{where } x_p \text{ is a new variable.} \end{array}$$

It follows from the definitions that σ satisfies (1). Furthermore, as σ maps distinct variables to distinct variables, we have

$$\mathrm{inst}(u_{(\exists \mathcal{A}_p\ \phi)}) = \mathrm{inst}(u_{(\exists \mathcal{A}_p\ \phi)}\sigma) = \mathrm{inst}(\{u_{(\exists \mathcal{A}_p\ \phi)}\sigma[u/x_p] \mid u \in \mathcal{T}(\mathcal{F}, \mathcal{X})\}).$$

We conclude by combining this equality with (1). □

The following property of the projection operation is the last argument required for justifying our implementation of projection:

Property 3 *For every term u and every path p,*

$$\exists p\ \mathrm{inst}(u) = \mathrm{inst}(\exists p\ u).$$

We now establish the main soundness lemma.

Lemma 5 *Let p be a path and ϕ a disjunction of closed conjunctions. Then*

$$\exists p\ [\![\phi]\!] = [\![\exists \mathcal{A}_p\ \phi]\!].$$

Proof 6 *We first prove the result in the case of a conjunction. The proof follows from the previous lemmas:*

$$\begin{array}{rcll} \exists p\ [\![\phi]\!] & = & \exists p\ \mathrm{inst}(u_\phi) & \textit{Lemma 3} \\ & = & \mathrm{inst}(\exists p\ u_\phi) & \textit{Property 3} \\ & = & \mathrm{inst}(u_{\exists \mathcal{A}_p\ \phi}) & \textit{Lemma 4} \\ & = & [\![\exists \mathcal{A}_p\ \phi]\!] & \textit{Lemma 3.} \end{array}$$

The proof for general case follows from the distributivity of projection, existential quantification, and meaning with respect to disjunction. □

5 Closing formulas

We have described a domain of formulas for which the projection is easily expressed as an existential quantification. However, this domain is not closed under conjunction. Therefore, to obtain a domain which is closed under union, intersection, and projection, we have to enforce the closure property on formulas while preserving their meaning. This is achieved by repeatedly "applying" the rules until they are all satisfied. This idea is slightly complicated by the fact that, to make the process terminate, we need an additional notion of *monotone* formula.

5.1 Rule application

The basic idea in order to make a conjunction satisfy a rule is to add some conclusion of the rule to the conjunction, if all the premises of the rule are in the conjunction. It is clear that this operation may make some other rules unsatisfied, therefore an iterative process is required, which we describe in Section 5.2.

5.1.1 Applying a rule to a conjunction

Given a rule $R \in \mathcal{R}$, and if

$$R = \frac{h_1 \ \ldots \ h_n}{c_1 \ \ldots \ c_m},$$

we define a formula $\mathrm{app}(\mathrm{R}, \phi)$ as the result of applying R to a conjunction ϕ:

$$\mathrm{app}(\mathrm{R}, \phi) = \begin{cases} \bigvee_{\substack{i \leq m \\ (\neg\, c_i) \notin \phi}} (\phi \ \wedge \ c_i) & \text{if } \forall i \leq n \ h_i \in \phi \\ \phi & \text{otherwise.} \end{cases}$$

The application of a rule to a conjunction clearly makes this rule satisfied, without changing the meaning of the conjunction. Now we need to extend this definition to formulas. We do this using the DNF representation, thus we have to show that the obtained formulation is independent of the particular expression of a formula. Furthermore, we will eventually have to compute it with basic boolean function operations. For these two reasons we need a more "semantic" definition of rule application, which is given in Lemma 6. This definition requires a notion of *monotone* conjunction, and in particular a special operator on boolean functions used to ensure monotony.

5.1.2 Making conjunctions monotone

We say that a conjunction ϕ is *monotone* with respect to a set of atoms $A \subseteq \mathcal{A}$ if it does not contain any negated atom $\neg\, a$ with $a \in A$. A conjunction is *monotone* if it is monotone with respect to $\mathcal{A}$. Note that we use the term *positive* for a distinct purpose in this report.

Given a set of atoms A, we define function $\mathrm{monotone}_A : \mathcal{B} \to \mathcal{B}$ whose purpose is to make conjunctions monotone with respect to A, while preserving monotony with respect to any other positive atoms. $\mathrm{monotone}_A$ is defined by the following boolean expression (only the existence of this expression matters):

$$\mathrm{monotone}_A(\phi) = \left(\exists A \ \left(\phi \ \wedge \ \bigwedge_{a \in A} a \implies a' \right) \right) [a/a' \mid a \in A]$$

where for all $a \in A$, a' is a fresh boolean variable, and $[a/a' \mid a \in A]$ is the substitution which renames each a' to a. The only thing that matters about the above expression is that it only uses standard logic operations, namely, conjunction, existential quantification, and renaming.

5.1.3 Applying a rule to a DNF

Lemma 6 gives a propositional expression of the application of rule to a conjunction.

Lemma 6 *Let ϕ be a monotone conjunction and $R \in \mathcal{R}$, and let*

$$R = \frac{h_1 \ \ldots \ h_n}{c_1 \ \ldots \ c_m}.$$

Then

$$\mathrm{app}(\mathrm{R}, \phi) = \mathrm{monotone}_{\{h_1, \ldots, h_n\}}(\phi \ \wedge \ \phi_R)$$

where ϕ_R is defined as

$$\phi_R \ = \ h_1 \ \wedge \ \cdots \ \wedge \ h_n \implies c_1 \ \vee \ \cdots \ \vee \ c_n.$$

In this expression, the function $\text{monotone}_{\{h_1,\ldots,h_n\}}$ has no incidence on the meaning of the computed formula, which suggests that rule application could have been defined just as a conjunction with R. The role of monotony is to ensures that only the consequence of R on the particular formula ϕ are kept, which is essential for the rule propagation process to terminate.

Given Lemma 6, and given that $\wedge$ and monotone_A are both distributive with respect to $\vee$, we can safely extend the application function to arbitrary formulas. This is done in Lemma 7.

Lemma 7 *Let $\phi = \bigvee_{i\in I} \phi_i$ be a disjunction of monotone conjunctions and $R \in \mathcal{R}$, and define $\{h_1, \ldots, h_n\}$ and ϕ_R as in Lemma 6. Then*

$$\text{monotone}_{\{h_1,\ldots,h_n\}}(\phi \ \wedge \ \phi_R) = \bigvee_{i\in I} \text{monotone}_{\{h_1,\ldots,h_n\}}(\phi_i \ \wedge \ \phi_R).$$

We denote this formula by $\text{app}(\text{R}, \phi)$ *and we have*

$$\text{app}(\text{R}, \phi) = \bigvee_{i\in I} \text{app}(\text{R}, \phi_\text{i}).$$

Proof 7 *The first point is a consequence of the distributivity of $\wedge$ and* monotone_A *with respect to $\vee$, the latter being itself a consequence of the distributivity of conjunction, existential quantification, and renaming. The second point follows from Lemma 6.*

Example 4 *Consider again the conjunction $\phi = \epsilon \mapsto f \ \wedge \ f.1 = f.2 \ \wedge \ f.2 = f.3$ of Example 3, as well as a second conjunction $\psi = \epsilon \mapsto f \ \wedge \ f.1 \mapsto a \ \wedge \ f.2 = f.3$. The disjunction of the two represents the set of terms whose form is one of $f(u, u, u)$ or $f(a, u, u)$. Suppose that we want to apply to $\phi \ \vee \ \psi$ the rule R corresponding to* TRANS *for the equalities $f.1 = f.2$ and $f.2 = f.3$. We first take the conjunction with ϕ_R, which yields the following DNF:*

$$\begin{array}{rcl} (\phi \ \vee \ \psi) \ \wedge \ \phi_R & = & \epsilon \mapsto f \ \wedge \ f.1 = f.2 \ \wedge \ f.2 = f.3 \ \wedge \ f.1 = f.3 \\ & & \vee \ \epsilon \mapsto f \ \wedge \ f.1 \mapsto a \ \wedge \ f.1 = f.3 \\ & & \vee \ \epsilon \mapsto f \ \wedge \ f.1 \mapsto a \ \wedge \ f.1 \neq f.2. \end{array}$$

Then we apply the function $\text{monotone}_{\{f.1=f.2, f.2=f.3\}}$ *which discards the dis-equality $f.1 \neq f.2$ in the third disjunct, and we obtain the desired result:*

$$\begin{array}{rcl} \text{app}(\text{R}, \phi \ \vee \ \psi) & = & \epsilon \mapsto f \ \wedge \ f.1 = f.2 \ \wedge \ f.2 = f.3 \ \wedge \ f.1 = f.3 \\ & & \vee \ \epsilon \mapsto f \ \wedge \ f.1 \mapsto a. \end{array}$$

The second disjunct is discarded, as it is less general than $\epsilon \mapsto f \ \wedge \ f.1 \mapsto a$.

Formally, Lemma 8 shows how the function app can be used to enforce the satisfaction of rules in (the disjuncts of) a formula, while keeping the formula monotone and prefix-closed, and preserving its meaning.

Lemma 8 *Let ϕ be a disjunction of monotone, prefix-closed conjunctions, and $R \in \mathcal{R}$. Then*

- $[\![\text{app}(\text{R}, \phi)]\!] = [\![\phi]\!]$*, and*
- $\text{app}(\text{R}, \phi)$ *can be expressed as a disjunction of monotone, prefix-closed conjunctions which satisfy R as well as the rules satisfied by ϕ whose premises do not occur in the conclusion of R.*

Proof 8 *We first prove the result for the case where ϕ is a single monotone, prefix-closed conjunction. In this case, the fact that $[\![\text{app}(\text{R}, \phi)]\!] = [\![\phi]\!]$ follows from the definition of* app*, the definition of meaning, and Property 1. The second point is easily checked from the definitions.*

In the general case, the result follows from the second point of Lemma 7 and the distributivity of meaning with respect to disjunction (Lemma 1).

5.2 Iterating rules

Procedure 1 is used to compute the closure of a formula with respect to $\mathcal{R}$. It uses a simple work-set algorithm which keeps track of the set of rules that may still need to be applied. Rules are applied one by one, following a particular strategy: they are selected by increasing length of the longest path appearing their atoms. This strategy ensures the termination of rule propagation (see Lemma 10).

The partial correctness of Procedure 1 is proved in Lemma 9.

Lemma 9 *Let ϕ be a disjunction of monotone, prefix-closed conjunctions. If* $\text{closure}(\phi)$ *terminates then*

procedure closure(ϕ) =
  Let $w = \{R \in \mathcal{R} \mid R$ has a premise in the support of $\phi\}$
  while $w \neq \emptyset$ **do**
    choose a rule $R \in w$ with a minimum max path length
    $\phi \leftarrow \mathrm{app}(\mathrm{R}, \phi)$
    $w \leftarrow w \setminus \{R\}$
    if ϕ has changed **then**
      $w \leftarrow w \cup \{R' \in \mathcal{R} \mid$ some premise of R' is a conclusion of $R\}$
  return ϕ

Procedure 1: Closure of ϕ with respect to $\mathcal{R}$

- $[\![\mathrm{closure}(\phi)]\!] = [\![\phi]\!]$, *and*
- $\mathrm{closure}(\phi)$ *can be expressed as a disjunction of monotone, closed conjunctions.*

Proof 9 *We prove that the closure algorithm maintains the following invariant:* ϕ *can be expressed as a disjunction of monotone, prefix-closed conjunctions which satisfy every rule not in* w*, and* $[\![\phi]\!]$ *is equal to its initial value.*

- *Initially, this is clear from the hypotheses, and from the fact that any rule whose premises are not in the support of a conjunction is satisfied by this conjunction.*
- *Assume that a disjunction* ϕ *satisfies the invariant for some set* w *of rules. Let* $R \in w$*. Then Lemma 8 ensures that* $\mathrm{app}(\mathrm{R}, \phi)$ *has the same meaning as* ϕ *and that its disjuncts satisfy* R *and the rules of* w *whose premises do not occur in the conclusion of* R*. Furthermore, if* $\mathrm{app}(\mathrm{R}, \phi) = \phi$*, then its disjuncts obviously satisfy all the rules not in* w *by hypothesis. We conclude that the invariant is maintained after one iteration of the loop.*

At the end, we conclude from the fact that $w = \emptyset$*, since monotony implies positiveness.*

The termination of Procedure 1 is stated in Lemma 10.

Lemma 10 *Let* ϕ *be a disjunction of monotone, prefix-closed conjunctions. Then* $\mathrm{closure}(\phi)$ *terminates.*

Idea of the proof. Let $\bigvee_{i \in I} \phi_i$ be a DNF of the initial formula to which Procedure 1 is applied. In the execution of the algorithm, the computed formula ϕ can always be expressed as a disjunction of conjunctions where each disjunct is the result of applying the rules selected so far to some ϕ_i. Thus, the algorithm can only diverge if for some ϕ_i the repeated application of the rules yields an infinite set of atoms, and therefore a set of atoms with arbitrary long paths. This implies that $[\![\phi_i]\!] = \emptyset$, and by Proposition 2 we conclude that $\phi_i \vdash_{\mathcal{R}} \bot$. As the rules are selected by increasing path length, and since rules with arbitrary long paths are selected, those implied in this deduction will eventually be selected as long as they apply and the conjunction be removed, which contradicts the hypothesis.

5.3 The domain of monotone closed formulas

We have defined in Section 3.2 a kind of boolean functions to express sets of terms. We have given in Sections 4.2 and 4.4 sufficient constraints that allow the projection operation on sets of terms defined in Section 4.1 to be expressed as an existential quantification on formulas. Finally, we have shown in Sections 5.1 and 5.2 how to restore the only one of those constraints which is not preserved by the conjunction operation (namely, the satisfaction of the rules in $\mathcal{R}$), at the cost of an additional *monotony* requirement. Together, those results make possible the definition of a boolean function-based domain for representing and computing on sets of terms, as follows.

Definition 1 *The domain* $\mathcal{D} \subseteq \mathcal{B}$ *is defined as the set of formulas that can be expressed as disjunctions of monotone, closed conjunctions.*

Theorem 1 states that $\mathcal{D}$ can represent the set of instances of a term, is closed under disjunction, quantification with respect to $\mathcal{A}_p$ for any p, and conjunction followed by closure, and that these operations compute respectively the union, projection on p, and intersection operators on the meaning of formulas.

Theorem 1 *Let* u *be a term and* $\phi, \phi' \in \mathcal{D}$*. Let* p *be a path and* $\mathcal{A}_p$ *defined as in Lemma 4. Then the following holds.*

$$\begin{array}{rcrcl}
\phi_u & \in \mathcal{D} \text{ and} & [\![\phi_u]\!] & = & \mathrm{inst}(u) \\
\phi \vee \phi' & \in \mathcal{D} \text{ and} & [\![\phi \vee \phi']\!] & = & [\![\phi]\!] \cup [\![\phi']\!] \\
\exists \mathcal{A}_p\ \phi & \in \mathcal{D} \text{ and} & [\![\exists \mathcal{A}_p\ \phi]\!] & = & \exists p\ [\![\phi]\!] \\
\mathrm{closure}(\phi \wedge \phi') & \in \mathcal{D} \text{ and} & [\![\mathrm{closure}(\phi \wedge \phi')]\!] & = & [\![\phi]\!] \cap [\![\phi']\!]
\end{array}$$

Proof 10 *The fact that $\phi_u \in \mathcal{D}$ follows from the definitions. By definition, $\mathcal{D}$ is also obviously closed by disjunction. Given the distributivity of existential quantification, it is easy to check from the definitions (in particular the definitions of A_p and $\mathcal{R}$) that $\exists \mathcal{A}_p\ \phi\ \in \mathcal{D}$. Finally, $\phi \wedge \phi'$ can clearly be expressed as a disjunction of monotone, prefix-closed conjunctions. We conclude by applying Lemmas 1, 2, 5, 9, and 10.*

6 Experiments

We have implemented our ideas in a prototype solver based on the `cudd` BDD library [23]. This solver is built from the same basic elements which constitute the theory that we have presented, but with a few optimisations.

Most importantly, the boolean function encoding is slightly different: we relax the monotony constraint used to ensure the termination of the closure procedure, in order to obtain better BDD performance. The reason is that the mutual exclusion of function symbols for a given path cannot be represented in monotone formulas, which authorises spurious boolean valuations and causes an explosion of the number of BDD nodes. Instead, the termination of the closure procedure is enforced by using an additional set of atoms, which intuitively express whether the sub-term reached by a given path is "materialised" or not, and by only ensuring the monotony with respect to these atoms. We believe that this idea would also be useful to enable negation in our formalism, as it introduces negated atoms without compromising the termination of rule propagation.

The implementation of the immediate consequence operator T_c, it is mostly based on the naive formulation described in Section 2 (adapted to the use of $\exists p$ instead of π_p), but we apply some well-known optimisations to avoid as much as possible the large intermediate relations. One of them is the early projection of the sets of matching substitutions (substitutions are restricted to the variables that appear in the head or in subsequent premises). Another idea is the use of path renaming, boolean identification of atoms, and a limited use of intersection, instead of the costly intersection with the term $\langle u, \mathit{subst}(x_1, \ldots, x_p)\rangle$ of Section 2.

Other optimisations include the use of regular types to reduce the number of atoms, small changes in the choice of rules, and a less conservative initial work-set in the rule propagation process.

The fix-point iteration procedure which computes the least model of a logic program is a standard work-set algorithm with a simple strategy based on the dependency graph between clauses, and in particular its strongly connected components. The termination of the computation over boolean functions does not directly follow from the finiteness of the least fix-point, since the representation of a set of terms is not unique and we do not provide a decision procedure for inclusion. We conjecture that testing implication between boolean functions ensures termination in the same cases, and we have not found any counter-example.

The least fix-point, restricted to a particular (small) predicate of interest, is converted back to a set of terms using Lemma 3, which allows us to (implicitly) enumerate the terms from a disjunctive normal form of the formula. We apply it to the *minimal* DNF, which can be computed efficiently by iterating over BDD nodes rather than BDD paths (which correspond to a non-minimal DNF). We believe that the disjuncts of the minimal DNF for any formula in $\mathcal{D}$ can be shown to satisfy the hypotheses of Lemma 3, which makes this strategy sound.

Case studies

We have applied our prototype to simple test cases, such as the n-queens and the dining philosophers problems. Some sets of terms obtained as solutions of our examples exhibit an efficient BDD encoding, which we measure by the fact that the order of magnitude of the number of nodes in the BDD is significantly smaller than the number of most general terms in the corresponding set. For example, the representation of the state space and transition relation for the 14 dining philosophers (which has 228486 states and 2067856 transitions) involves BDDs whose size does not exceed 20000 nodes with intermediate relations staying under 400000 nodes throughout the computation. However, the obtained performance is not sufficient yet: the execution time (27 hours for the above example) and even the memory usage are often much higher than with an explicit representation of terms, for example with XSB. In particular, we have encountered critical *variable ordering* issues. So far, our treatment of variable ordering relies on general-purpose heuristics which are part of the `cudd` package, but a late scheduling of reordering causes an explosion in the BDD size. Some formulas also remain large even after reordering, which suggests that our encoding can be further improved.

7 Related work

A large range of existing and ongoing research is relevant to the goal of the present work, most notably on the topic of logic program compilation. As for the original proposal of this report, which is a boolean function-based domain (targeted towards BDDs), connections exist with other data structures which have been proposed for representing sets of terms in various contexts, and with different properties. In the following, we compare our work with two of those.

7.1 Term indexing

A lot of research about theorem provers has been concerned with various *term indexing* techniques [15], which are representations (or abstractions) of sets of first-order terms. Those data structures are designed to be compact and to offer fast but generally approximate answers to some particular queries, such as the retrieval of a given term, or set of terms. Some of those representations, *tree-based* indexing techniques, even provide union and unification of sets of terms (the latter of which is equivalent to our intersection operation). In particular, the *adaptive discrimination tree* [22] representation is very similar to our boolean function encoding, if boolean functions are represented with BDDs. In those trees, nodes are labelled by paths (similar to the paths that we used here), and edges are labelled with function symbols to be matched with the path labelling their source nodes. Thus the structure is the same as that of a BDD with only "function symbol" atoms. The most obvious difference is that there is no sharing between sub-trees as in (reduced) BDDs. *Substitution tree indexing* [14] is a more general technique which is able to exactly represent sets of terms. The treatment of variable equality is different from ours: variable are represented in a concrete way, while we treat each equality as a single atom. Still, we are not aware of any term indexing technique which could represent sets of terms in an exact way, and have a much smaller size than the number of terms.

7.2 Tree automata

Another data structure for representing sets of first-order terms is *tree automata* [11]. In their simplest form, tree automata are able to represent the domain of *regular* sets of terms, which is incomparable with the domain considered here, *i.e.*, sets of instances of finite sets of terms. On one hand, tree automata can express the nesting of a given pattern any number of times, which yields an infinite number of most general terms. The representation is rather different from BDDs, as a term corresponds to a partial unfolding of the automata, *i.e.*, a tree, while in BDDs a term corresponds to a BDD path, *i.e.*, a word. On the other hand, tree automata recognise sub-terms independently, thus they are not able to express sub-term equality. Various extensions of tree automata with equality or dis-equality constraints have been proposed. The *most general class* [11, Chapter 4], which allows the transitions to be constrained by arbitrary boolean combination of path equalities and dis-equalities, subsumes our propositional formula domain. Unfortunately, the computations that we study are not feasible with this most general class of tree automata with equalities and dis-equalities: emptiness is undecidable even for less expressive sub-classes. It is decidable for some of those however, such as *automata with constraints between brothers* [6] and *reduction automata* [12, 10]. We believe that only the latter is able to represent the instances of a finite number of arbitrary terms. This class is closed under union and complementation, but we are not aware of an implementation of a projection operator for it.

8 Conclusions and further work

We have established the semantic foundations for computing with sets of first-order terms in the world of boolean functions. This opens the way for BDD-based manipulations of logic objects, and in particular, for a BDD-based bottom-up execution of logic programs. As a conclusion, we discuss the main motivation for such a goal, which is static program analysis, then we list a few possible directions for pursuing our quest for expressiveness.

8.1 Application to static analysis

First, we should stress that the BDD-based solver that we are building is not intended as a general-purpose Prolog engine. In the following we try to recount the path which led to the combination of BDDs and logic programs for implementing scalable static program analyses, in particular Control Flow Analysis.

8.1.1 Logic languages and static analysis

The idea of computing the result of a static analysis as the least fix-point of a logic program is at the core of abstract compilation [16], which is a popular static analysis technique among the logic programming community. It consists in abstracting a logic program p by another logic program $p^\sharp$. The result of the analysis of p is obtained by running the program $p^\sharp$. In [1], Albert *et al.* added one more compilation layer. For analysing Java bytecode programs, they compiled them into equivalent Prolog programs that were thereafter themselves analysed. In the Control Flow Analysis domain, some of the current authors proposed to use Datalog for expressing context-sensitive analyses of object-oriented programs [4].

8.1.2 Using BDDs to solve logic programs

An even closer inspiration for the current study was the work of Whaley and Lam [26] who proposed the BDD-based DATALOG engine `bddbddb` as a framework for designing scalable context-sensitive static analyses in a very natural declarative fashion. An off-the-shelf logic solver such as CORAL [20] or XSB [21] could of course have been used to solve the DATALOG clauses, and in fact, the XSB system (which implements a tabulation mechanism allowing to compute the well-founded semantics of logic programs) has been used on a small scale in a program analysis context for computing groundness and strictness information [13]. The reason for designing a dedicated BDD-based solver is that data computed by static analyses is unusual. First, the amount of computed data is huge, and second, so is the amount of redundancy, which seems to be exactly the application range for BDDs.

8.1.3 Introducing first-order terms

In Whaley and Lam's work however, the lack of expressiveness of DATALOG is perceptible, as it fails to represent a key aspect of the analysis *viz.*, the context sensitive call graph. For this crucial step, the authors hand-craft an *ad hoc* algorithm maximising the sharing properties of BDDs. On the other hand, the availability of terms in our solver allows a straightforward representation of calling contexts by lists of callers, with a simple logic specification, and whose BDD encoding features similar sharing. Furthermore, this enables an easy experimentation of various choices of abstractions for calling contexts. Overall, our contribution to the field of analysis specification by logic programs is the added expressiveness of first-order terms.

8.1.4 Lifting the "range-restricted" limitation

In [24, Chapter 5], we have used boolean formulas to represent finite sets of ground terms, which is enough to compute the semantics of *range restricted* programs (*i.e.*, every variable appearing in the head of a clause must also appear in the body). This case was significantly simpler, because no equality atoms were required, and the formulas could just be "materialised" up to given *depth*, following the maximum term depth. A new difficulty here is that a common materialisation depth cannot be found for all the terms of a formula, and must instead be chosen at the conjunction level, which is achieved through the rule propagation mechanism described in Section 5.

8.2 Perspectives on expressiveness

A natural direction for future research is to investigate the specification of static program analyses in PROLOG, and in particular the possible benefits of terms such as lists and trees, of which we have given an example with the calling contexts in context-sensitive analyses. On the other hand, we can try to accept an even more expressive class of programs with two main ideas.

8.2.1 Magic transformations

The semantics that we have presented here computes the least fix-point of definite logic programs, but using an infinite union. So, the actual computation only terminates if the least fix-point is finite, *i.e.*, can be expressed as the set of instances of a finite number of terms. This is a significant limitation to the programs that we can accept: many standard recursive predicates, such as list operations (*e.g.*, append), do not have a finite least fix-point.

This situation can be greatly improved by applying *magic transformations* [3] to the programs. The idea of these automatic program transformations is to specialise the predicates such that they only compute terms which contribute to a given query. The transformed program somehow simulates a top-down execution of the original one from the specified query. Many standard infinite predicates can be eliminated by applying magic transformations, and this is what we have done (manually) for our case studies. In the future, we will automate this treatment.

Finally, even if applied to finite predicates, magic transformations may still be useful for performance reasons, as they make the least fix-point smaller. In particular, for the application to static analysis, we believe that a demand-driven implementation could be obtained almost for free by using this technique.

8.2.2 Stratified negation

In this report we have considered *definite programs*, *i.e.*, without negation. However, a bottom-up semantics can be given to logic programs with *stratified* negation, *i.e.*, where the negation is used only between the strongly connected components of the dependency graph, but is not involved in cyclic dependencies. To introduce stratified negation in our formalism, the only operation needed is the complementation of sets of terms. This operation is straightforwardly implemented by a negation of formulas, but the resulting formulas escape from the domain that we used, as they are not positive (and even less monotone). Overall, dealing with negation requires an adaptation of most of our results, but we believe that these difficulties may be

overcome, and that our solver can be extended to accept any logic program with stratified negation (and with a finite least fix-point). From the static analysis point of view, this covers most of the uses of negation, due to the monotony properties of such analyses.

Acknowledgements The authors thank Bertrand Jeannet for his help on BDDs, and Delphine Demange and David Pichardie for their comments on this report.

References

[1] Elvira Albert, Miguel Gómez-Zamalloa, Laurent Hubert, and Germán Puebla. Verification of Java bytecode using analysis and transformation of logic programs. In *Proc. of the 9th International Symposium on Practical Aspects of Declarative Languages*. Springer, 2007.

[2] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of the 29th symposium on Principles of programming languages (POPL '02)*. ACM, 2002.

[3] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. of the 5th symposium on Principles of database systems (PODS'86)*. ACM, 1986.

[4] Frédéric Besson and Thomas Jensen. Modular class analysis with Datalog. In *Proc. of the 10th Static Analysis Symposium*. Springer-Verlag, 2003.

[5] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Simple and efficient relational querying of software structures. In *Proc. of the 10th Working Conference on Reverse Engineering (WCRE'03)*. IEEE Computer Society, 2003.

[6] Bruno Bogaert and Sophie Tison. Equality and disequality constraints on direct subterms in tree automata. In *STACS '92: Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 1992.

[7] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Compututers*, 35(8), 1986.

[8] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and Lain-Jinn Hwang. Symbolic model checking 10^{20} states and beyond. *Information and Computation*, 98(2), 1992.

[9] Gianpiero Cabodi and Marco Murciano. BDD-based hardware verification. In *Proc. of the 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*. Springer, 2006.

[10] Anne-Cécile Caron, Hubert Comon, Jean-Luc Coquidé, Max Dauchet, and Florent Jacquemard. Pumping, cleaning and symbolic constraints solving. In *ICALP '94: Proceedings of the 21st International Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1994.

[11] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007.

[12] M. Dauchet, A.-C. Caron, and J.-L. Coquidé. Reduction properties and automata with constraints. *Journal of Symbolic Computation*, 20, 1995.

[13] Steven Dawson, Coimbatore R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Proc. of the conference on Programming language design and implementation (PLDI '96)*. ACM, 1996.

[14] Peter Graf. Substitution tree indexing. In *RTA '95: Proc. of the 6th International Conference on Rewriting Techniques and Applications*. Springer-Verlag, 1995.

[15] Peter Graf. *Term Indexing*. Springer-Verlag, 1996.

[16] Manuel V. Hermenegildo, Richard Warren, and Saumya K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13, 1992.

[17] A. J. Hu, D. L. Dill, A. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *Proc. of the 4th Int. Workshop on Computer Aided Verification (CAV'92)*. Springer-Verlag, 1993.

[18] Mizuho Iwaihara and Yusaku Inoue. Bottom-up evaluation of logic programs using binary decision diagrams. In *Proc. of the 11th International Conference on Data Engineering (ICDE'95)*. IEEE Computer Society, 1995.

[19] David B. Kemp, Peter J. Stuckey, and Divesh Srivastava. Magic sets and bottom-up evaluation of well-founded models. In *Proc. of the 1991 Int. Symposium on Logic Programming*. MIT, 1991.

[20] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the coral deductive database system. In *SIGMOD '93: Proc. of the 1993 international conference on Management of data*. ACM, 1993.

[21] Prasad Rao, Konstantinos Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A system for efficiently computing well-founded semantics. In *Logic Programming and Non-monotonic Reasoning*, 1997.

[22] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *ICALP '92: Proc. of the 19th International Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1992.

[23] Fabio Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, 2009. Available on: http://vlsi.colorado.edu/ fabio/CUDD/.

[24] Tiphaine Turpin. *Pruning program invariants*. PhD thesis, Université de Rennes 1, December 2008.

[25] John Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, March 2007.

[26] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the conference on Programming language design and implementation (PLDI '04)*. ACM Press, 2004.

A The dining philosophers

We reproduce below the logic program corresponding to our "dining philosophers" example. Our language features regular types, but it should be self explanatory. Note the use of the `subReachable` predicate, which ensures the termination of recursive predicates on lists.

```
type philosopher =
  | thinking
  | hasLeftFork
  | eating

type fork =
  | free
  | used

type element = {
  philosopher : philosopher;
  rightFork : fork
}

type table = element list

predicate initial(element)
initial({philosopher = thinking ; rightFork = free}).

predicate reachable(table)
reachable([P1, P2, P3]) :- initial(P1), initial(P2), initial(P3).

predicate subReachable(table)
subReachable(T) :- reachable(T).
subReachable(T) :- subReachable([_ | T]).

predicate transition(table, table)
predicate specialTransition(table, table)

transition([{philosopher = P ; rightFork = free}, {philosopher = thinking ; rightFork = F} | T],
           [{philosopher = P ; rightFork = used}, {philosopher = hasLeftFork ; rightFork = F} | T]).

transition([{philosopher = hasLeftFork ; rightFork = free} | T],
           [{philosopher = eating ; rightFork = used} | T]).

transition([{philosopher = P ; rightFork = used}, {philosopher = eating ; rightFork = used} | T],
           [{philosopher = P ; rightFork = free}, {philosopher = thinking ; rightFork = free} | T]).

transition([P | T], [P | T1]) :- subReachable([P | T]), transition(T, T1).

predicate takeLastFork(table, table)
takeLastFork ([{philosopher = P ; rightFork = free}], [{philosopher = P ; rightFork = used}]).
takeLastFork([P | T], [P | T1]) :- subReachable([P | T]), takeLastFork(T, T1).

predicate releaseLastFork(table, table)
releaseLastFork ([{philosopher = P ; rightFork = used}], [{philosopher = P ; rightFork = free}]).
releaseLastFork([P | T], [P | T1]) :- subReachable([P | T]), releaseLastFork(T, T1).

specialTransition([{philosopher = thinking ; rightFork = F} | T],
                  [{philosopher = hasLeftFork ; rightFork = F} | T1]) :- takeLastFork(T, T1).
specialTransition([{philosopher = eating ; rightFork = used} | T],
                  [{philosopher = thinking ; rightFork = free} | T1]) :- releaseLastFork(T, T1).

reachable(T1) :- reachable(T), transition(T, T1).
reachable(T1) :- reachable(T), specialTransition(T, T1).

predicate waiting(element)
waiting({philosopher = hasLeftFork ; rightFork = used}).

predicate stuck(table)
stuck([]).
stuck([P | T]) :- subReachable([P | T]), waiting(P), stuck(T).

predicate deadLock(table)
deadLock(T) :- reachable(T), stuck(T).
```